Introduction to Cross Memory Services

By P. R. Dorn,
Washington Systems Center

U. Pimiskern
WTSC Poughkeepsie

DAPS Code 0894
GG22-9201-00
June 1980

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either express or implied. The use of this information is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment.

Requests for copies of this and other IBM publications should be made to your IBM representative or to the branch office serving your locality.

A form is provided in the back for any comments or criticisms you may have. If the form has been removed, comments may be addressed to IBM Washington Systems Center, Field Support, 18100 Frederick Pike, Gaithersburg, MD 20760 Comments become the property of IBM.

Washington Systems Center
Gaithersburg, Maryland
Technical Bulletin

Introduction to Cross Memory Services

P. R. Dorn
U. Pimiskern
(WTSC - Poughkeepsie)

DAPS Code 0894
GG22-9201-00
June 1980

ABSTRACT

This Technical Bulletin contains an overview presentation of the cross memory architectural extensions provided by MVS/System Product Release 2 (MVS/SP R2). This presentation will discuss the following:

- Why cross memory significantly enhances the current System/370 architecture,
- How MVS and its subsystems will use cross memory services,
- The functions provided by the new cross memory services.

However, this presentation will not discuss the details of the architecture or services provided by this enhancement.

There are actually two MVS/SP R2 products:

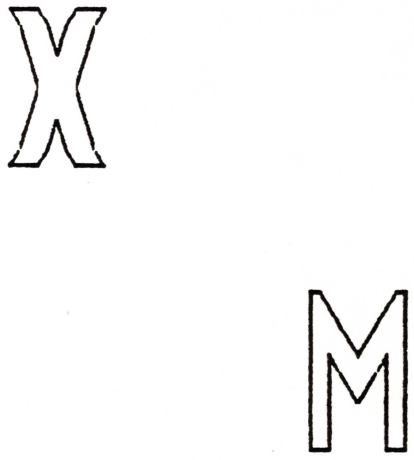
1. MVS/System Product-JES2 Release 2 (5740-XYS), and
2. MVS/System Product-JES3 Release 2 (5740-XYN).

The primary difference between the two products is the subsystem supplied (either JES2 or JES3). Unless otherwise indicated, the terms "MVS/System Product Release 2" and "MVS/SP R2" will be used to describe both products.

THIS PAGE INTENTIONALLY LEFT BLANK

PRESENTATION

The following foils and accompanying text provide an an overview of the cross memory architectural extensions provided by MVS/System Product Release 2.


X
M

FOIL 1

This presentation provides an overview of the cross memory architectural extensions provided by MVS/System Product Release 2 (MVS/SP R2). This product currently has a planned general availability date of June 1981. This presentation will discuss the following:

- Why cross memory significantly enhances the current System/370 architecture,
- How MVS and its subsystems will use cross memory services,
- The functions provided by the new cross memory services.

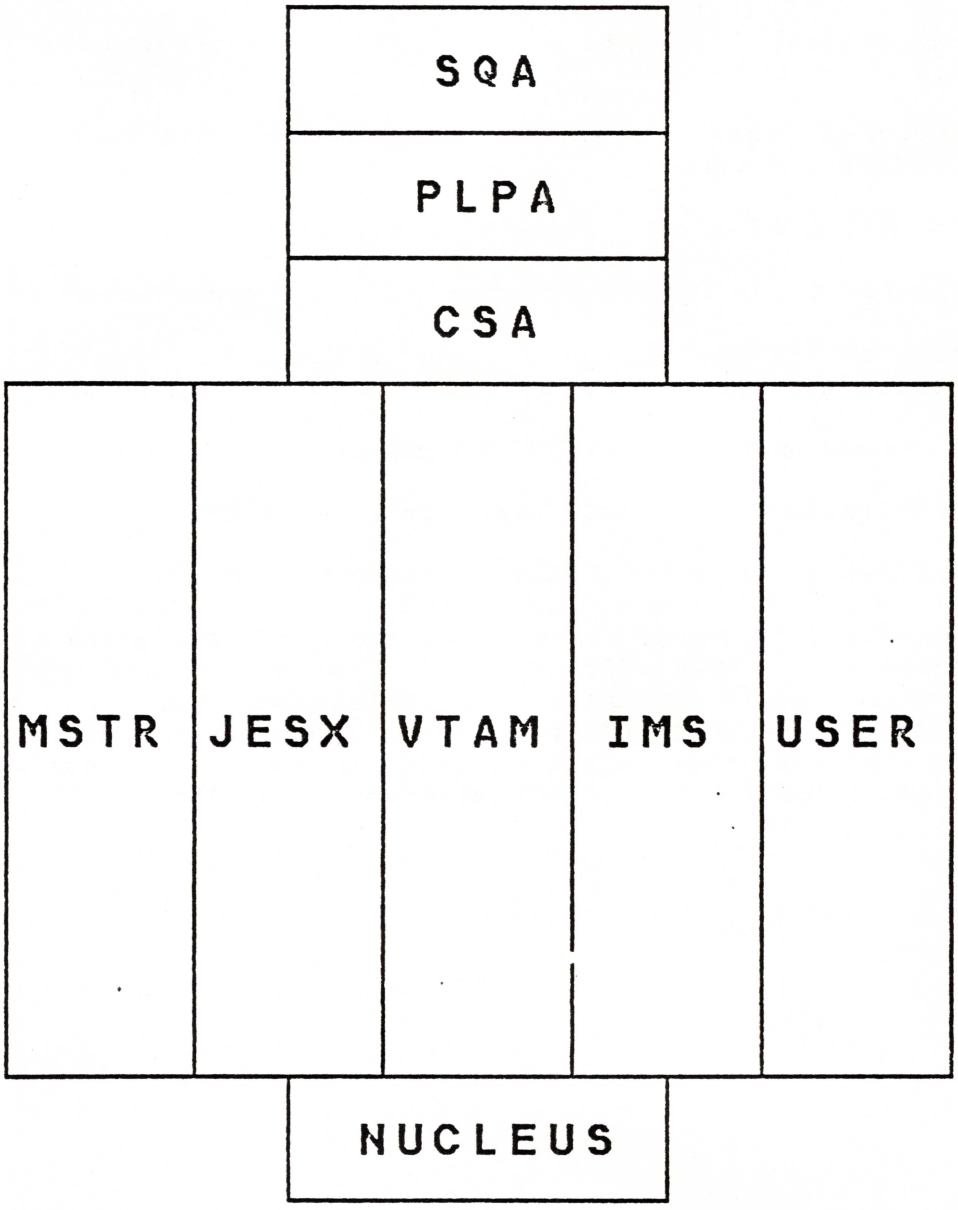
However, this presentation will not discuss the details of the architecture or services provided by this enhancement.

There are actually two MVS/SP R2 products:

1. MVS/System Product-JES2 Release 2 (5740-XYS), and
2. MVS/System Product-JES3 Release 2 (5740-XYN).

The primary difference between the two products is the subsystem supplied (either JES2 or JES3). Throughout this presentation, the terms "MVS/System Product Release 2" and "MVS/SP R2" are used to describe both products. Where one product is different than the other (e.g., describing the use of cross memory services by JES3), the particular product is identified.

CLASSICAL VIRTUAL STORAGE MAP



IBM WASHINGTON SYSTEMS CENTER
FOIL 2

FOIL 2

Here we see the "standard" storage map used to describe MVS. Note that there are two types of areas in this map:

- Areas addressable from all address spaces (e.g., Nucleus, CSA), and
- Unique areas only addressable from a single address space (e.g., the private area).

Also, one can see from the diagram that the private area is suppose to be approximately the same size as the common area.

ADVANTAGES OF HAVING MULTIPLE PRIVATE ADDRESS SPACES

- CAPACITY
  - EFFECTIVELY "INFINITE" NUMBER OF ADDRESS SPACES
  - MULTI-ADDRESS SPACE SUBSYSTEMS (E.G., IMS)
- SYSTEM INTEGRITY
  - CONTAINMENT OF ERRORS (EXCEPT COMMON)
  - LOCALIZATION OF ERRORS AND RECOVERY

FOIL 3

This multiple address space concept provides a significant improvement over previous IBM operating systems (e.g., MVT and SVS). First of all, it provides a major increase in the virtual storage capacity of the system. Single level address schemes such as SVS require that the **sum** of all the users' virtual storage requirements be less than 16 megabytes. Assuming the installation wants to run multiple jobs concurrently, this probably means the largest single user is limited to two or three megabytes. Also, several large jobs can not execute concurrently.

The current architecture will support over 1,500 concurrent users, with each user (address space) having a large region of several megabytes.[1] Even with the large growth in computing power, this limit should last us for quite a while.

The size of the private area is not the upper bound on the amount of virtual storage available to an application. For example, IMS/VS[2] can utilize up to 32 different address spaces (31 dependent regions and 1 control region). Even assuming a maximum private area of 5 megabytes, this means that IMS/VS can acquire 160 megabytes of private virtual addressing.

The multiple address space concept also improves system "Reliability, Availability and Serviceability" (RAS). Previous systems ensured that application programs were unable to destroy key system data. However, privileged routines (both IBM and customer supplied) are given access to all of virtual storage. Programming errors occasionally cause the accidental overlaying of critical control blocks, thus forcing a re-IPL of the system.

With MVS, even system routines are not given access to all of virtual storage. Therefore, if the program cannot address the storage, it cannot accidentally destroy it. Unfortunately, some of the critical control blocks are still in commonly addressable storage (e.g., SQA). and can be destroyed.

Since many control blocks are in private storage (e.g., TCBs), localization of errors is enhanced. For example, destroying the TCB chain typically results in the termination of a single address space rather than the entire system.

---

1 Because of other limitations (e.g., amount of paging space, real memory), MVS cannot actually create this many address spaces. However, from a virtual addressing perspective, it is feasible.

2 Information Management System, 5740-XX2.

HOWEVER --

HAVE INTER-ADDRESS SPACE COMMUNICATION PROBLEMS

- PROGRAM SHARING
- DATA MOVEMENT
- DATA SHARING

FOIL 4

Although MVS provides significant advantages over single level addressing systems, it does introduce a new requirement: **the ability to communicate among address spaces**. For example, communication between the IMS/VS control region and its dependent regions is much more difficult under MVS than under SVS or VS1.

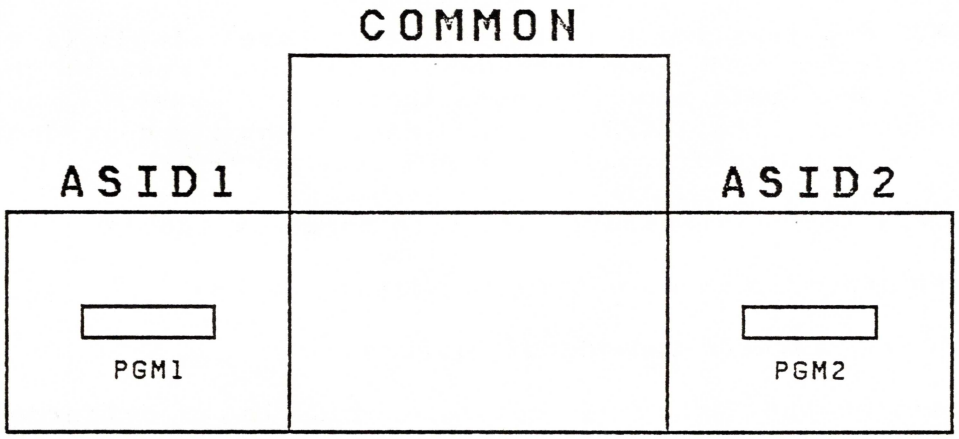
The communication requirements fall into three categories:

1. Sharing programs among multiple address spaces,

2. Moving data between two address spaces, and

3. Referencing and updating data located in a different address space without having to move the data to common storage.

MULTI-ADDRESS SPACE ACCESS



- PGM1 WANTS TO:
  - CALL PGM2
  - MOVE DATA FROM PGM2
  - MOVE DATA TO PGM2
  - ACCESS DATA IN PGM2
- COMMUNICATION REQUIRES:
  - COMMON STORAGE (CSA)
  - SERVICE MANAGEMENT

FOIL 5

This foil shows the inter-address space communication required to support multi-address space subsystems.[3] To provide full inter-address space communication, a program (PGM1) executing in address space 1 (ASID1) must be able to:

- Execute a program which resides in a different address space.
- Move data residing in its address space to another address space.
- Move data residing in another address space to its own address space.
- Manipulate data contained in another address space using the standard System/370 instruction set (e.g., Load Halfword, Store, Edit and Mark).

Obviously, several of the functions are related. For example, if PGM1 "calls" PGM2 residing in another address space, PGM2 must be able to access the data in the calling address space (ASID1). Similarly, PGM2 may need to move data from ASID1 to its own address space, and return the answers to ASID1.

To accomplish this inter-address space communication currently requires:

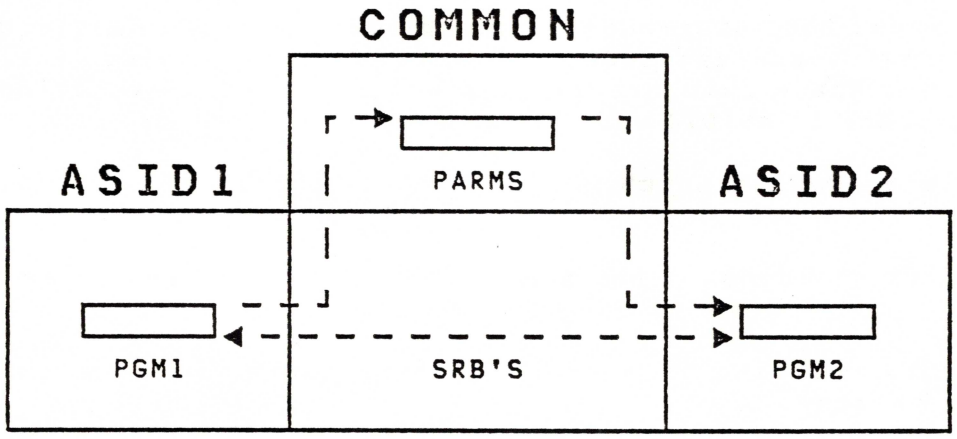
1. Commonly addressable storage (e.g., CSA or SQA), and
2. Service Management (e.g., SRB Schedule, Cross Memory Post).

Next we will look at techniques that have been developed by MVS subsystems to support inter-address space communication.

---

[3] Unless otherwise indicated, the word "subsystem" does not refer to the formalized definition (e.g., JES2, JES3), but to a major component such as IMS/VS, ENQ/DEQ, or VTAM.

PROGRAM SHARING TECHNIQUE



1 SCHEDULE SRB TO ASID2

2 WAIT

3 SRB EXECUTES PGM IN ASID2

4 POSTS CALLER (SRB)

5 POST ENDS

6 PGM RESTARTS

FOIL 6

Here we see a program residing in ASID1 needing to execute a program in ASID2. Normally, the program in ASID2 is shared by multiple address spaces. For example, a program in the IMS/VS control region may be used by all the dependent IMS/VS regions. These shared programs typically reside in the subsystem address space.

To execute the program in ASID2, the following normally occurs:

1. PGM1 constructs and schedules an SRB to execute in ASID2. This includes finding the address of the routine to call.

2. PGM1 must now issue a WAIT (or SUSPEND) so the SRB can execute.

3. MVS then dispatches the SRB in ASID2. Note that this SRB gains control at the priority of ASID2 and is dispatched with a higher priority than the TCB related work in ASID2. This means that processing for ASID1 (as well as other calling address spaces) is often performed at a different (and usually higher) dispatching priority. This technique obviously reduces the ability of address space priority to control performance.

4. Before terminating, the SRB must inform the calling program (PGM1) that it is complete. Normally, PGM2 will use a branch entry cross memory post to perform this notification. The branch entry is required since an SRB cannot issue SVCs.

5. The cross memory post causes an SRB to be scheduled in ASID1. This SRB will "wake up" PGM1.

6. PGM1 now resumes execution.

PROBLEMS WITH THIS TECHNIQUE

- SYSTEM OVERHEAD OF MULTIPLE SRBS
- PGM LINKAGE VIA AUTHORIZED FUNCTION
- COMPLEX TO DESIGN/PROGRAM
  - USER PROVIDES OWN SYNCHRONIZATION
  - SRB PROCESSING DIFFICULT
  - RECOVERY CONSIDERATIONS
- PASSING PARAMETERS
  - DATA IN SRBPARM
  - DATA IN COMMON

FOIL 7

From the previous discussion, it should be clear that using SRBs to share programs is not straight forward. We required 2 SRBs and multiple passes through the dispatcher to accomplish the sharing. Also, SRBs can only be scheduled while in key-0, supervisor state. This means that the calling program must be APF[4] authorized or the subsystem must provide an SVC to actually perform the SRB scheduling. This adds to the system overhead required to perform the program sharing.

As you can imagine, this interface is difficult to use. The previous example assumes everything proceeds without errors. It does not address the following concerns:

- If the SRB abends, who cleans up any outstanding work, and who notifies PGM1 that the SRB is complete (and has failed).

- If the called address space disappears, what happens when a user tries to schedule an SRB to it.

Up until now we have assumed the program executes in a vacuum. That is, PGM1 does not need to pass input data to PGM2, nor does it expect results. There are very few programs which meet either of these criteria. There are two ways to pass data between address spaces:

1. Place the parameter(s) in SRBPARM. Unfortunately, SRBPARM only allows for 32 bits of information. Also, this field can only be used for input.

2. Place the parameters in commonly addressable storage (e.g., CSA).

Most programs are forced to do the latter.

---

[4] Authorized Program Facility

THEREFORE, PRODUCTS -

- PLACE THEIR MODULES IN PLPA
  - REDUCES THE SIZE OF THE PRIVATE AREA
- REPLICATE THEIR CODE IN EVERY ADDRESS SPACE
  - INCREASES REAL STORAGE REQUIREMENTS OF THE SYSTEM
  - INCREASES I/O LOAD

FOIL 8

To bypass these problems, most IBM and customer packages do one of the following:

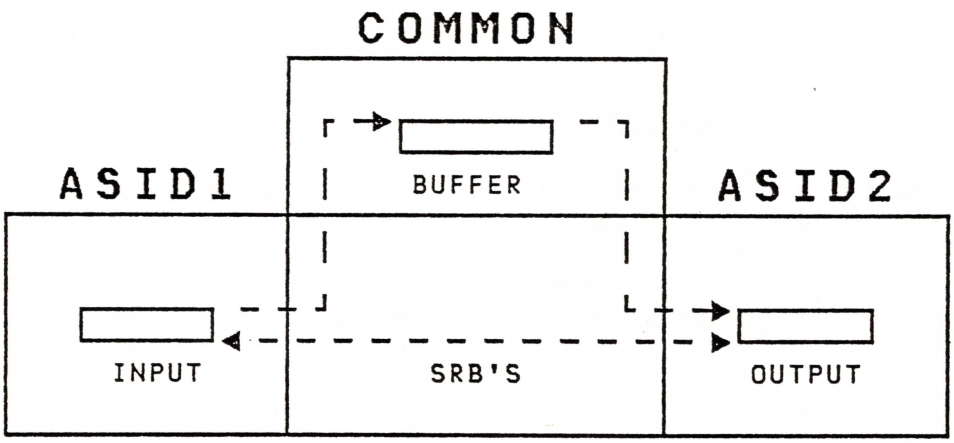
1. Place all their modules in PLPA.

   Since the modules reside in commonly addressable storage, they are accessible from all address spaces. However, this reduces the amount of storage available in each user's private area; even in those applications not using the function. Ideally, only heavily used modules should be placed in PLPA.

2. Replicate their code in every address space.

   Rather than place the module in PLPA, some subsystems force a copy of the module to be loaded into every address space using it. While this reduces the size of PLPA, it increases the real storage load on the system since multiple copies of the module reside in storage. It also increases the load of the paging subsystem since each copy of the module is swapped in and out of storage. Finally, it reduces the effectiveness of the processor's lookaside buffer since the different versions of the module translate to different real addresses.

DATA MOVEMENT TECHNIQUE



1 MOVE DATA
TO COMMON

2 SCHEDULE
SRB TO ASID2

3 WAIT

4 SRB MOVES DATA
FROM COMMON

5 POST ASID1
(SRB)

6 POST ENDS

7 PGM RESTARTS

IBM WASHINGTON SYSTEMS CENTER
FOIL 9

FOIL 9

Here we see a program residing in ASID1 needing to move data to ASID2. Since the data cannot be moved directly, the program must first move the data to common storage (usually CSA). Then it must schedule an SRB to execute in ASID2. Since the SRB executes asynchronously, the program must suspend execution via a WAIT macro. The system then dispatches the SRB in ASID2. The SRB moves the data from common to the private address space. After the move is complete, the SRB must post the program in ASID1. This cross memory post schedules an SRB to ASID1 which actually posts the ECB. Then the program in ASID1 can resume execution.

PROBLEMS WITH THIS TECHNIQUE

- DATA PASSES THROUGH COMMON
  - INTEGRITY EXPOSURES
  - NO CONTAINMENT OF ERRORS
  - SECURITY CONCERNS
- SLOW DATA MOVEMENT
- DATA MOVEMENT VIA AUTHORIZED FUNCTION
- COMPLEX TO DESIGN/PROGRAM
  - USER PROVIDES OWN SYNCHRONIZATION
  - SRB PROCESSING DIFFICULT
  - RECOVERY CONSIDERATIONS

FOIL 10

This data movement technique has the same problems as the program sharing technique previously described. In fact, the two techniques are almost identical. However, in addition to the previously described problems, one must now be concerned with the integrity and security of the data while residing in common.

If the data contains sensitive information, the data must be fetch protected. Also, the subsystem probably wants to protect the data from accidental and/or malicious update by a user program. This means the data must reside in key 0 through 7. While this protects the data from alteration by unauthorized programs, it does not preclude accidental destruction of the data by authorized programs. For example, if running an IMS/VS production and an IMS/VS test system in the same processor, there is the possibility of the IMS/VS test system destroying the key 7 production data contained in CSA.

THEREFORE, PRODUCTS --

- LEAVE ALL THEIR DATA IN CSA
  - CREATES A LARGE CSA REQUIREMENT
  - REDUCES PRIVATE AREA SIZE
- USE A SINGLE ADDRESS SPACE
  - CREATES A LARGE PRIVATE AREA REQUIREMENT
  - SYSTEM CONTROLS ARE NOT EFFECTIVE
  - AP/MP PERFORMANCE CONCERNS

FOIL 11

To bypass these problems, most IBM and customer packages do one of the following:

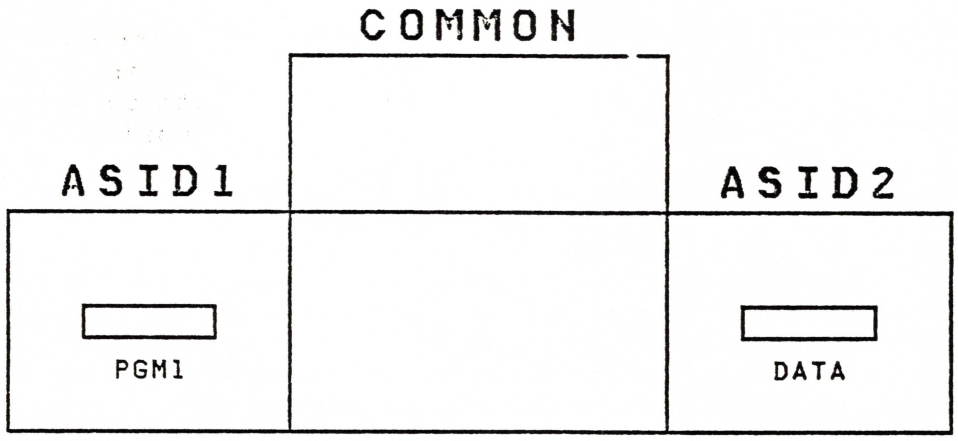
1. Keep their data in CSA

   This technique obviously increases the size of CSA and reduces the size of the user's private area. If executing two copies of the subsystem (e.g., a production and a test version), the amount of CSA needed is the total of the two subsystem requirements. However, no two address spaces need to reference both portions (test and production) of CSA. In fact, most installations would prefer that no address space could reference both portions.

2. Use a single address space

   By using only a single address space, a product can ignore the data movement problem. However, this technique creates a very large private area problem. Also, since MVS provides control on an address space basis, subsystems using a single address space cannot be controlled by the installation using system provided options (e.g. dispatching priority, multi-programming level, priority I/O queuing). Finally, a single address space typically cannot take full advantage of an AP or MP configuration. Usually contention within an address space significantly reduces the effectiveness of the second CPU.

DATA SHARING TECHNIQUE



NO WAY TO ACCOMPLISH WITH CURRENT ARCHITECTURE

FOIL 12

To provide full inter-address space communication, one needs the capability to reference and update data located in a different address space without having to move the data to common storage. The current MVS architecture does not support a technique for accomplishing this data sharing.

MULTI-ADDRESS SPACE PROBLEMS

- INDIRECT COMMUNICATION
  - EXCESSIVE USE OF COMMON
  - SRB OVERHEAD
  - COMPLEXITY
- AUTHORIZATION
  - SYSTEM PROVIDES SINGLE LEVEL (APF)
  - SUBSYSTEMS MUST PROVIDE THEIR OWN

FOIL 13

Thus multi-address space subsystems currently have two major deficiencies:

1. The inability to efficiently communicate among address spaces.

   As we have seen, inter-address space communication is difficult to program and inefficient to use. Therefore, most subsystems use common storage (PLPA, CSA and SQA) to contain their programs and data.

2. No selective authorization.

   MVS provides a single level of authorization, namely APF (Authorized Program Facility). Therefore, subsystems typically provide their own authorization schemes (usually employing a subsystem supplied SVC). What is actually required is the ability to provide two authorization organizations:

   a. Vertical

      In this hierarchy, some address spaces have more authority than others. For example, the IMS/VS control region has more authority than its dependent regions.

   b. Horizontal

      In this organization, some address spaces have different (but equal) authority than others. For example, the IMS/VS control region for an IMS/VS test system needs equal authority to the control region for an IMS/VS production system. However, the test control region should only be able to communicate with its dependent regions, while the production control region should only be able to communicate with its dependent regions.

MVS TODAY

- PLPA LARGE AND GROWING
- CSA/SQA LARGE AND GROWING EVEN FASTER
- APPLICATION PROGRAM REQUIREMENTS GROWING
  - MOST PROGRAMS SMALL
  - A FEW NEED LARGE DATA AREAS
- REACHING THE LIMIT OF 16 MEGABYTES OF VIRTUAL STORAGE

FOIL 14

Thus we see that MVS today is starting to reach some of the limits imposed by the current S/370 architecture. The size of the common areas is growing at a rapid rate. This is particularly true for CSA. It takes a long time to develop a megabyte of code, but very little time to generate a megabyte of data.

We also have application programs requiring more private area storage. While most programs require very little storage (probably less than a megabyte), many customers have several critical programs requiring large amounts of data (e.g., FORTRAN structural analysis programs).

With both common and private area requirements growing, moving data between common and private does not alleviate the situation. Only an expansion of the S/370 addressing architecture provides the required growth path.

Therefore --

A SOLUTION --

CROSS MEMORY
SERVICES
&
ARCHITECTURAL
EXTENSIONS

FOIL 15

With MVS/System Product Release 2, MVS supports an extension to the architecture that provides facilities for improved linkage and communication between MVS address spaces. Use of cross memory services by major system components and subsystems (e.g., JES3 and IMS/VS) offers the potential to reduce system virtual storage requirements and improve data isolation.

It is anticipated that the majority of jobs and users will realize the benefits of cross memory through their usage of the system components and subsystems that use cross memory services, rather than modifying customer programs to use the cross memory services directly.

As previously indicated, there are actually two MVS/SP R2 products:

1. MVS/System Product-JES2 Release 2 (5740-XYS), and

2. MVS/System Product-JES3 Release 2 (5740-XYN).

The primary difference between the two products is the subsystem supplied (either JES2 or JES3). Throughout this presentation, the terms "MVS/System Product Release 2" and "MVS/SP R2" are used to describe both products. Where one product is different than the other (e.g., describing the use of cross memory services by JES3), the particular product is identified.

CROSS MEMORY SERVICES

PROVIDE --

- DIRECT COMMUNICATION AMONG ADDRESS SPACES
  - PROGRAM SHARING
  - DATA MOVEMENT
  - DATA SHARING
- LEVELS OF AUTHORIZATION

FOIL 16

Cross memory services address the two major problems with multi-address space subsystems:

1. Inefficient inter-address space communication

   As we have previously seen, there are three types of inter-address space communication:

   a. Sharing programs among multiple address spaces,

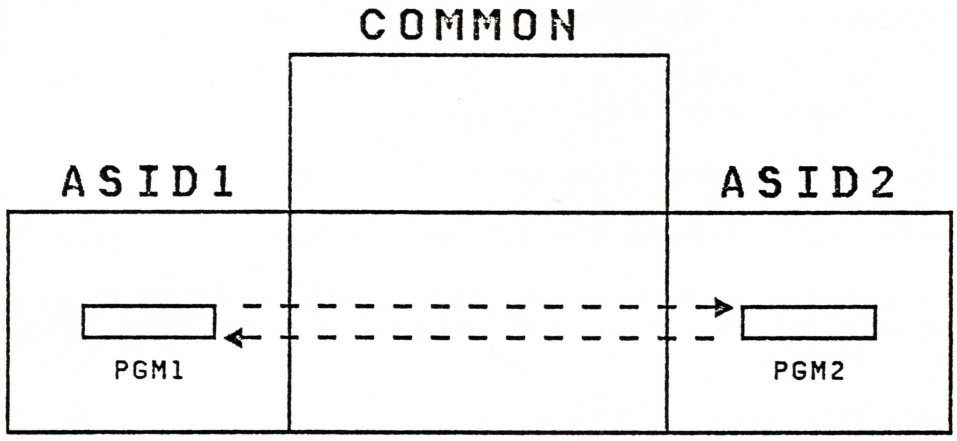
   b. Moving data between two address spaces,

   c. Referencing and updating data located in a different address space, without having to move the data to common storage.

2. Lack of authorization levels

   Currently, MVS provides only one level of authorization (APF). Vertical and horizontal levels of authorization would provide additional security and integrity to multi-address space subsystems.

PROGRAM SHARING MECHANISM



1 "CALL" ASID2 PGM2 — ADDRESSING ASID1 & ASID2

2 EXECUTE PGM2

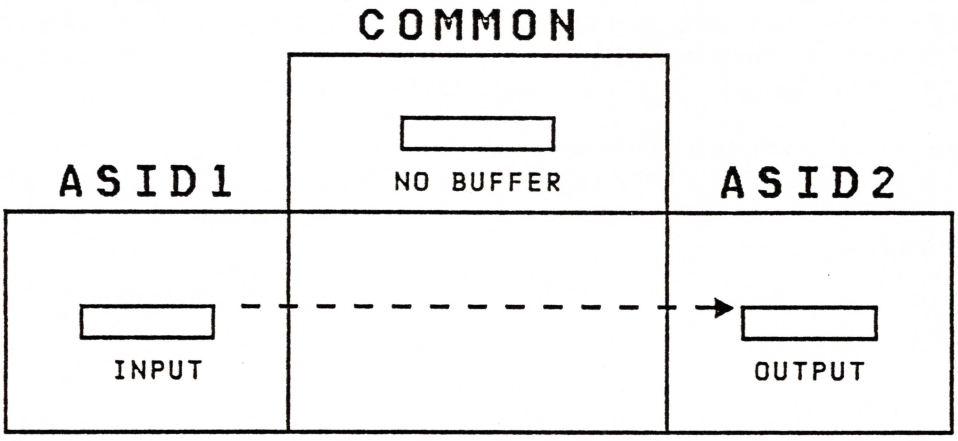
3 "RETURN"

FOIL 17

Here we see how cross memory services support program sharing among address spaces. PGM1 in ASID1 "calls" PGM2 which resides in ASID2. When PGM2 receives control, it is able to access data in both ASID1 and ASID2. Note that this "call" is synchronous; that is, PGM1 does not continue to execute. PGM2 then executes. When PGM2 is complete, it "returns" to PGM1 in ASID1.

This scenario accomplishes the same function as shown in foil 6 on page 12. This "call"/"return" sequence replaces an SRB schedule, a WAIT, and a cross memory POST (which implies another SRB schedule).

DATA MOVEMENT MECHANISM



1 EXTEND
ADDRESSING
TO ASID2

2 MOVE DATA

FOIL 18

Cross memory services also improve inter-address space data movement. With MVS/System Product Release 2, the following scenario permits data to be moved between address spaces:

1. Tell MVS to extend addressing to ASID2.

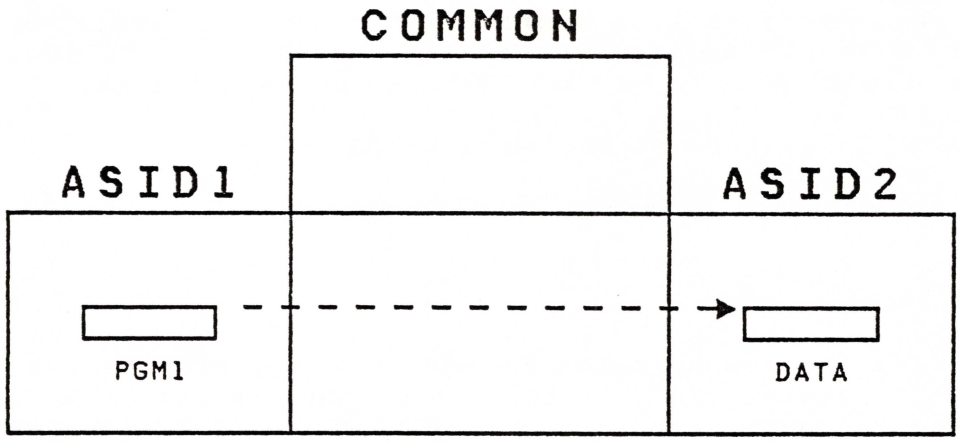
   The user can now reference/update data in ASID2 as well as ASID1.

2. Move the data.

   The data can now be moved between the two address spaces without passing the data through common. In this example, we see the data being moved from ASID1 to ASID2. The data could have just as easily been moved in the other direction.

This process should be compared to foil 9 on page 18 which demonstrates the technique currently used to pass data between address spaces. This cross memory move eliminates (1) an SRB schedule, (2) a WAIT, and (3) a cross memory POST (which creates another SRB schedule). This cross memory move also (4) eliminates the requirement for a buffer in common storage.

DATA SHARING MECHANISM



1 ESTABLISH
ADDRESSING
TO ASID2

2 SWITCH DATA
ADDRESSING

3 EXECUTE S/370
INSTRUCTIONS

UPDATE/REF
ASID2 DATA

4 RETURN TO
STANDARD
ADDRESSING

FOIL 19

With cross memory services, one can now execute instructions using the data from another address space. This provides a capability which was not previously available with MVS. The following sequence provides multi-address space data sharing:

1. PGM1 in ASID1 uses cross memory services to provide addressing to ASID2.

2. The user then tells MVS and the processor that data references are to use ASID2 (e.g., the fullword updated by the "ST" instruction).

3. When finished with the data in ASID2, PGM1 can return to its normal addressing mode (i.e., data accesses using ASID1).

SUPPORTED PROCESSORS

- ALL MACHINES WITH S/370 EXTENDED FACILITY
- OPTIONAL EXTENSION FEATURE FOR 3033 (INCLUDING 3033N) - HARDWARE & MICROCODE
- OPTIONAL 3031 MICROCODE EC

FOIL 20

Cross memory services are provided for all processors supported by MVS/System Product Release 2. MVS/SP R2 supports the following processors:

Model 158-1	UP/MP/AP
Model 158-3	UP/MP/AP
Model 168-1 + RPQ S20579	UP/MP
Model 168-1 + RPQ S20580	AP
Model 168-3	UP/MP/AP
3031 Processor	UP/AP
3032 Processor	UP
3033 Processor	UP/MP/AP
3033 Model Group N	

MVS/SP R2 Supported Processors

MVS/SP R2 requires the appropriate System/370 Extended Feature on System/370 Models 158 (feature 7730 or 7731) and 168 (feature 7730).

Cross memory is also implemented by an optional 3033 Extension feature (number 6850) on the 3033 processor. This architectural extension, implemented in hardware/microcode, will enhance MVS/SP R2.

An optional microcode engineering change (EC) will also be available for the 3031 processor.

Cross memory services will be supported in the guest operating system environment under the level of the VM/System Product (5664-167) which is most current at the time of availability of MVS/System Product Release 2. However, the cross memory services will execute under the VM/SP without using the optional 3033 Extension feature or the optional 3031 Engineering Change.

PERFORMANCE INFORMATION

TO BE AVAILABLE 4Q80

FOIL 21

MVS/System Product Release 2 performance data is currently not available. It will, however, be provided fourth quarter of 1980.

TYPICAL USES

- DATA-ONLY ADDRESS SPACES
  - KEEP LARGE ARRAYS
  - KEEP DATA USED BY MULTIPLE ADDRESS SPACES
  - DATA ISOLATION
- PROGRAM EXECUTION
  - EXTENSION OF PLPA
  - PROGRAM "CALLED" BY ANOTHER ADDRESS SPACE
  - PROGRAM ACCESSES ANOTHER ADDRESS SPACE'S DATA
- ALL OF THE ABOVE

FOIL 22

Auxiliary address spaces are typically used for two purposes:

1. As a "data only" address space.

   That is, the address space contains data but no code. Other address spaces use cross memory services to reference and update the data contained in this auxiliary memory. This auxiliary memory could be used by only one address space or shared among several address spaces. The following are examples of the use of a data only auxiliary address space:

   - To keep large arrays used by a single address space
   - To keep data used by a multi-address space subsystem (e.g., JES3).
   - To protect the data from accidental destruction by the application program.

2. To contain executable code.

   In this mode, the auxiliary address space is used as an extension to PLPA. The programs in this address space can be used in two modes:

   - Via a "call" from another address space.
   - As a "normal" program under its own TCB. In this case, the program accesses the data contained in other address spaces.

Obviously, a subsystem can use an auxiliary address space to contain both code and data.

We will now look at some of the MVS components and subsystems which use cross memory services in MVS/System Product Release 2. We will not spend much time looking at the component function, but rather will concentrate on how the component exploits the cross memory architectural extensions.

GLOBAL RESOURCE SERIALIZATION
(ENQ/DEQ ENHANCEMENTS)

- CONTROL BLOCKS KEPT IN THIS ADDRESS SPACE
- QUERY ACCESS TO CNTL BLKS
- ENHANCED SYSTEM RAS
- QCB'S AND QEL'S NO LONGER IN SQA
- EXECUTES UNDER CALLER'S AND ITS OWN TCB

FOIL 23

Global resource serialization enhances the current ENQ/DEQ mechanism by:

1. Extending the ENQ/DEQ function to the multiple processor environment, thus providing the ability to serialize resource access across processor boundaries.

2. Replacing the current ENQ tables (chained QCBs and QELs) with tables that are accessed via a hashing technique.

3. Moving the ENQ/DEQ blocks from SQA to a new cross memory address space.

When the user issues an ENQ (or DEQ), the SVC will "call" the new processing routine using cross memory services. This routine will update the control blocks contained in the new auxiliary address space. This new address space is listed as "GRS" by the DISPLAY command. The moving of the control blocks to a private address space provides the following benefits:

- Common virtual storage (i.e., SQA) is not required to contain the ENQ/DEQ blocks. This savings is most important to the large TSO environment (since the number of QCBs and QELs is normally directly related to the number of address spaces in the system).

- System RAS is significantly improved. Key-0 uses will no longer be able to accidentally destroy the ENQ/DEQ control blocks. **If you can't access the storage, you can't accidentally destroy it.**

Customer programs which scan the ENQ/DEQ control blocks in SQA will no longer work properly. MVS/System Product Release 2 provides a new system service, GQSCAN, to extract the required information from this new control block structure. Although conversion to GQSCAN will cause programming changes in some customer environments, this interface should reduce future customer impact. After MVS/SP R2, the entire ENQ structure could be changed without impacting the user. Only the results returned to GQSCAN would have to remain constant. The application program is now insulated from the internal structure of the global resource serialization component.

Global resource serialization is an example of an MVS component which uses an auxiliary address space to contain code and data. It also executes the code under the caller's TCB as well as its own TCB. The current ENQ/DEQ functions of this component execute under the caller's TCB, although they update control blocks in the global resource serialization address space. The new functions of global resource serialization (i.e., support of multiple processors) execute under a TCB located in the global resource serialization address space.

JES3

- STAGING AREAS
- PROTECTED BUFFERS
- REDUCED CSA REQUIREMENT
  - LOCAL AND GLOBAL PROCESSORS
- OPTIONAL (INISH DECK)

FOIL 24

JES3[5] has also been enhanced in MVS/System Product-JES3 Release 2 to optionally use cross memory services to significantly reduce its usage of common storage (CSA). With this level of JES3 the installation can specify that:

1. Secondary staging area extents are to be kept in a new auxiliary address space (listed as "JES3AUX" by the DISPLAY command).

2. USAM protected buffers are to be kept in this auxiliary address space.

This reduction applies to both GLOBAL and LOCAL JES3 processors.

Normally, JES3 CTC traffic requires very few staging areas. For many 3033 installations, 50K bytes will easily support this load. However, traffic occasionally backs up and additional staging areas are required to contain the messages until they can be transferred across the CTC. This back up is normally the result of:

1. Failure of the CTC, or

2. Failure of the JES3 GLOBAL processor.

Recovery of the I/O interface can often take fifteen to thirty or more minutes. Naturally, the number of staging areas required to contain these messages can be quite large (possibly 800K or more).

With MVS/SP R2, the installation can specify that these backup staging areas are to reside in the JES3AUX address space rather than in CSA. To transfer the data across the CTC, the staging area must reside in CSA. Therefore, data contained in the JES3AUX staging areas must be moved to the primary staging areas before actually being processed. However, if these secondary staging areas are infrequently used, there should be no noticeable system degradation. Thus in our example (which might represent a 3033UP), the installation could reduce its CSA staging area requirement by 800K.

The installation can also specify that all or part of the protected buffer pool (used to transfer data to the JES3 spool packs) be contained in the JES3AUX address space rather than in CSA. To support this feature, JES3 has added a new control block for each protected buffer (independent of whether the buffer is located in CSA or JES3AUX). This new block is 224

---

[5] Naturally this foil does not apply to MVS/System Product-JES2 Release 2.

bytes long, and resides in CSA. If an customer uses 240K bytes of 2K byte protected USAM buffers today, and decides to move all of them to an auxiliary address space after installation of MVS/SP R2, the CSA reduction will be:

(2K - 224) * (240K / 2K)

or

214K bytes

It should again be emphasized that this migration of staging areas and protected buffers to an auxiliary address space is optional. The use of the JES3AUX address space is indicated in the JES3 INISH deck.

THIS PAGE INTENTIONALLY LEFT BLANK

OPERATIONAL ENHANCEMENTS

- "DISPLAY" BLOCKS IN AUXILIARY ADDRESS SPACE
- ALLOCATION RECOVERY DATA IN AUXILIARY ADDRESS SPACE
- WOULD NOT BE IMPLEMENTED WITHOUT CROSS MEMORY

FOIL 25

Device allocation has been enhanced to maintain additional allocation information in a new auxiliary address space (known as "ALLOCAS"). For each device in the system, this auxiliary address space indicates all the users that are allocated to that device. This information is used for two purposes:

- "DISPLAY U" Command Enhancement

  The ALLOC parameter is used to request device allocation status on the "DISPLAY U" command. The supplied information will identify the address spaces (ASID and JOBNAME) allocated to the displayed device(s). This information can be particularly useful when a device must be unexpectedly removed from the system.

- Abnormal Address Space Termination

  When a DASD data set is unallocated, MVS decrements the count of the number of outstanding allocations to the associated device (maintained in the UCBUSER field of the UCB). However, when an address space abnormally terminates (e.g., paging error or FORCE command), normal unallocation is bypassed. This occasionally means that the UCBUSER field for a DASD UCB is not properly decremented. This will prevent the operator from varying the device offline (which requires UCBUSER to contain a value of 0).

  With MVS/SP R2, abnormal address space termination routines will use the data contained in the ALLOCAS address space to insure that the UCB allocation counter (UCBUSER) is properly maintained.

As one can imagine, the amount of data maintained in this auxiliary address space can be quite extensive. This is especially true in a large TSO environment. Without the cross memory services feature of MVS/SP R2, such a facility would have required enormous amounts of virtual common storage, and would have resulted in a considerable reduction in the size of the user's private area. In fact, the large amount of common storage required would have prevented these enhancements from being implemented.

SDUMP

- MEMORY USED TO BUFFER PAGEABLE, VOLATILE DATA TO AID SERVICABILITY
- DATA STORED IN AUXILIARY ADDRESS SPACE - NOT COMMON

FOIL 26

As part of SU33 (Dumping Improvements) SDUMP collects summary information about the failing user. This data includes, but is not limited to:

- The specified SUMLIST or SUMLISTA ranges.
- 2K of storage before and after each address contained in each register at the time of the failure.
- 2K of storage before and after the failing PSW address.

This data is normally gathered when the SDUMP is taken, thus preserving the data at the time of failure.

In certain cases (e.g., disabled or FRR routine), the actual dump cannot be taken until the user's cleanup routine executes. SDUMP currently supports a branch entry for use by any routine that is disabled, locked, or SRB mode. The branch entry causes a dump to be scheduled and control returned to the caller before the dump is complete. Unfortunately, the caller must often repair the data needed to solve the problem. Therefore the Summary Dump will show important data areas after they have been changed, rather than as they appeared when the error occurred.

Disabled Summary Dump solves this problem for disabled users by saving the volatile paged-in data prior to returning control to the SDUMP caller.

A new type of Summary Dump, the Suspend Summary Dump, is added with MVS/System Product Release 2. This function is meant for all branch entry callers of SDUMP who do not have global serialization (disabled). This volatile data is saved in a one megabyte buffer located in a new SDUMP address space (known as DUMPSRV). This is a "data only" auxiliary address space.

IMS USE OF CROSS MEMORY

- AVAILABLE JUNE 1981
- FUNCTIONALLY EQUIVALENT TO IMS 1.1.6 LSO OPTION
- ACCESSED THRU XM SERVICES:
  - SOME MODULES
  - DYNAMIC LOG BUFFERS
  - DYNAMIC LOGGER WORK AREA
  - GENERAL POOL
  - ISAM/OSAM BUFFER POOL
  - DISK LOG BUFFERS
  - LOGGER WORKAREA & BUFFERS
  - ENQ/DEQ CONTROL BLOCKS

FOIL 27

IMS/VS Version 1 Release 1.6, 5740-XX2, provides a new Local Storage Option (LSO) which reduces the amount of CSA used by IMS/VS. When the LSO facility is selected at system initialization, some IMS/VS modules will be loaded into the private area of the IMS/VS control region instead of into CSA. In addition, specific buffers and working storage areas are also moved from CSA to the IMS/VS control region private area. The size of each of these items is dependent upon parameters selected by the installation. The dependent region to control region communication is accomplished using SRBs.

Support enabling the local storage option facility to use cross memory services is planned for availability in June 1981. This support will be provided for the then most current release of IMS/VS. Thus, this LSO implementation replaces SRB scheduling with cross memory "direct" communication.

The following are moved from CSA to the IMS/VS control region in either implementation of the LSO option:

- Some IMS/VS modules. The current size of these modules is approximately 200K bytes.
- Dynamic Log Buffers
- Dynamic Logger Work Area
- Working Storage (General Pool)
- ISAM/OSAM Buffer Pool
- Disk Log Buffers[6]
- Logger Work Area and Buffers[6]
- ENQ/DEQ Control Blocks[6]

---

[6] For IMS/VS installations selecting the use of the Fast Path Feature, approximately 9K bytes of IMS/VS modules plus the indicated buffers and working storage are not moved from CSA to the control region private area.

OTHER USERS

- SELECTED USER APPLICATIONS
  - THOSE REQUIRING LARGE AMOUNTS OF DATA
  - THOSE REQUIRING MULTI-ADDRESS SPACE STRUCTURE

FOIL 28

IBM also realizes that certain customer programs could benefit from using the cross memory services provided by MVS/System Product Release 2. These programs fall into two categories:

1. Those requiring large amounts of data (e.g., large matrix manipulation programs).

2. Those applications requiring a multi-address space structure (e.g., RYO database programs).

However,...

NOT DESIGNED FOR --

MOST CUSTOMER PROGRAMS

(IF ANY)

FOIL 29

It is assumed that most customer programs will not be written to use cross memory services directly, but will realize the benefits of cross memory through their usage of the system components and subsystems that use cross memory services.

INTERFACE OVERVIEW

- CROSS MEMORY INTERFACES ARE PROBLEM PROGRAM ORIENTED
- "SUBSYSTEM" DEFINES AUTHORIZATION
- USES SUBSYSTEM APPROACH RATHER THAN SVC
- LATE BINDING OF USER TO THE SUBSYSTEM

FOIL 30

We will now spend time looking at the cross memory services provided by MVS/System Product Release 2. These cross memory interfaces are problem program oriented. That is, a program does not need any special authorization to request a cross memory function. However, as we will later see, the tables used by cross memory services can be constructed to restrict access to a particular class of users. This authorization is specified by the subsystem[7] that defines each cross memory access.

The cross memory interfaces are similar in design to the interfaces used by formalized subsystems (namely an externally defined table lookup). This approach is vastly superior to the SVC interface where the processing module, as well as the function definition of each SVC, is "hard coded" in the system. This external table look-up provides a late binding of the application program to the subsystem, which provides additional RAS benefits to the installation.

These preceding points will be elaborated on during the remainder of this presentation.

---

[7] Unless otherwise indicated, the word "subsystem" does not refer to the formalized definition (e.g., JES2, JES3), but to a major component such as IMS/VS, ENQ/DEQ, or VTAM.

SUBSYSTEM AUTHORIZES

ADDRESS SPACES WHICH --

- CAN "CALL" THE SUBSYSTEM ADDRESS SPACE(S) AT SPECIFIED ENTRY POINTS
- CAN REFERENCE/UPDATE THE SUBSYSTEM ADDRESS SPACE(S)
  - KEYS / FETCH-PROTECT STILL LIMIT STORAGE ACCESS

IBM WASHINGTON SYSTEMS CENTER
FOIL 31

FOIL 31

As we previously stated, the subsystem defines which address spaces can use each cross memory interface. The subsystem defines two types of authorization:

1. Which address spaces can "call" a subsystem address space at specified entry points. For example, IMS/VS may permit its dependent regions to "call" the control region at certain specified entry points. With this type of authorization, the dependent regions cannot branch to any other addresses in the control region.

2. Which address spaces can reference/update a subsystem address space. In this case, we are treating the subsystem address space as data. For example, IMS/VS could theoretically define that the control region can be accessed only by its dependent regions. Unlike the previous type of authorization, the dependent region can update/reference any address in the control region. Note, however, that permitting an address space to access another address space does not eliminate the standard key and fetch-protect mechanisms of the S/370 architecture. In our previous example, the dependent region would be able to access the control region, but not reference the data base buffers without switching to an appropriate storage key (0 or 7), assuming the dependent region is authorized to switch keys. Most subsystems will not permit any address spaces to access their data, but rather will authorize the subsystem to access their dependent address spaces.

SCOPE OF AUTHORIZATION

- ALL ADDRESS SPACES
- SELECTED ADDRESS SPACES
  - DIFFERENT ADDRESS SPACES CAN HAVE DIFFERENT AUTHORIZATION LEVELS
- CAN AUTHORIZE AN ADDRESS SPACE AT ANY TIME

IBM WASHINGTON SYSTEMS CENTER
FOIL 32

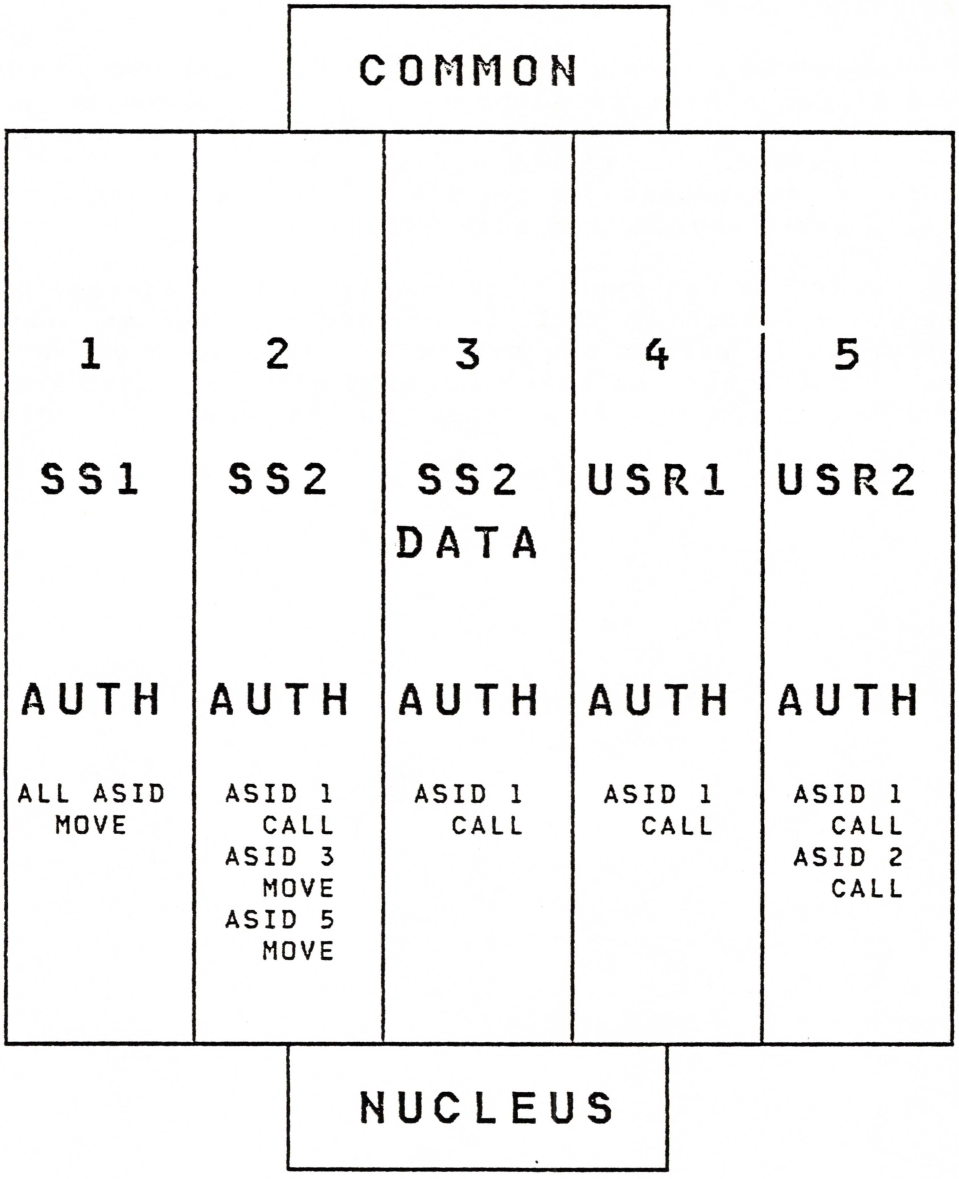
FOIL 32

There are two ways that the subsystem can authorize other address spaces to use cross memory services:

1. The subsystem can specify that all address spaces are given a particular level of authorization. For example, global resource serialization specifies that all address spaces are allowed to "call" its address space. With this technique, as new address spaces are created, they are automatically given the desired authorization.

2. The subsystem can specify that only certain address spaces are given authority. All other address spaces have no authority to access the subsystem. This type of authorization scope will be used by IMS/VS to insure that the control region can only be accessed by its dependent regions.

AUTHORIZATION EXAMPLE



FOIL 33

This is a sample MVS system showing some of the possible combinations of authorization. The following address spaces exist in this system:

- SS1 - a subsystem which can be "called" by all other address spaces. However, no one can update or reference the data in this address space. This implies that information in address space SS1 can only be accessed by "calling" the subsystem. The global resource serialization component is an example of this type of address space.

- SS2 - a subsystem which can only be "called" by associated address spaces (where "associated" is defined by the subsystem). A RYO data base manager might be an example of this type of subsystem.

- SS2 DATA - a data-only address space containing the data base used by the data base manager subsystem SS2.

- USR1 - an application program which is not authorized to use the data base subsystem.

- USR2 - an application program which is authorized to use the data base subsystem.

The following are typical authorizations that the individual subsystems would define:

- SS1

  Since this subsystem can be called by any address space, it probably must be capable of moving data to any address space.

- SS2

  This address space is authorized to "call" SS1 (which can be "called" by all address spaces). However, it cannot access data in SS1 (except perhaps by "query"). Subsystem SS2 must also be authorized to access data in its auxiliary data address space, "SS2 DATA." It probably must also be capable of accessing data in address spaces authorized to use this subsystem (e.g., USR2). Note, however, that it cannot access data in address spaces which cannot use the subsystem (e.g., USR1).

- SS2 DATA

  Even though this is a "data only" address space, it can theoretically "call" SS1. This phenomenon is the result of SS1 specifying that all address spaces can "call" it.

- USR1

  This address space is only authorized to "call" the universally available subsystem SS1. It is not authorized to "call" the data base manager subsystem SS2.

- USR2

  Like USR1, this address space can "call" subsystem SS1. However, it is also authorized to "call" the data base manager subsystem SS2. Note that USR2 cannot access the SS2 auxiliary address space. Only by "querying" the SS2 subsystem can USR2 reference or update the data base.

It should be noted that this is a trivial example of the authorization flexibility provided by cross memory services.

THIS PAGE INTENTIONALLY LEFT BLANK

EXTENSIONS TO "KEY"

- PROGRAM NOW GIVEN LIST OF KEYS AUTHORIZED TO USE
- MOST USERS ONLY ALLOWED KEY 8
- EXTENSION TO PSW KEY AND STORAGE KEY
- KEY LIST USED BY --
  - CHANGE KEY FUNCTION
  - "CALL" FUNCTION
  - "MOVE" FUNCTION

FOIL 34

Before preceding with a more detailed description of the cross memory services provided by MVS/System Product Release 2, we must look at the extensions provided to the System/370 key. A unit of work (e.g., TCB or SRB) is now given a list of keys that it is authorized to use. For example, a TCB may be authorized to use keys 5, 7 and 8. The application needs no special authorization to switch between these (e.g., supervisor state or APF authorized). Naturally, most user TCBs will only be authorized to use 1 key (which is contained in the TCBKEY field). This key value is normally 8. Users executing in supervisor state or APF authorized will still be allowed to switch to any key (0 through 15).

This key list is an extension of the PSW and storage keys that are defined in the current System/370 architecture. The system will still insure that fetch and store requests are permitted based upon the PSW and storage keys. The key list is not used in this authorization check. For example, a user authorized to use key 7, but currently executing with a PSW key of 8 will not be able to update key 7 storage using the standard S/370 instruction set.

This key list is used by:

- The previously described change key function
- The inter-address space "call" function
- The inter-address space "move" function

"CALL" FUNCTION

- PROCESSING IS TABLE DRIVEN
  (EXTERNAL TO USER)
- AUTHORIZATION CHECKING --
  - CALL ALLOWED
  - STATE
  - KEY(S)
- CAN CHANGE --
  - INSTRUCTION ADDRESS
  - ADDRESS SPACE
  - STATE
  - KEY(S) AUTHORIZED TO USE
  - ADDRESS SPACES MAY USE
  - PGMS AUTHORIZED TO CALL

IBM WASHINGTON SYSTEMS CENTER
FOIL 35

FOIL 35

We will now spend a few minutes looking at the "call" function. As previously indicated, this function is table driven, where the tables are external to the application program. When an application issues the "call" request, the system verifies that the request is authorized. This check insures that:

- The address space is allowed to issue the "call."
- The caller is in the correct state. For example, only supervisor state users may be able to request the desired function.
- The user is authorized to use a particular key. For example, only users who are authorized to use key 5 or 7 may be authorized to request the specified function.

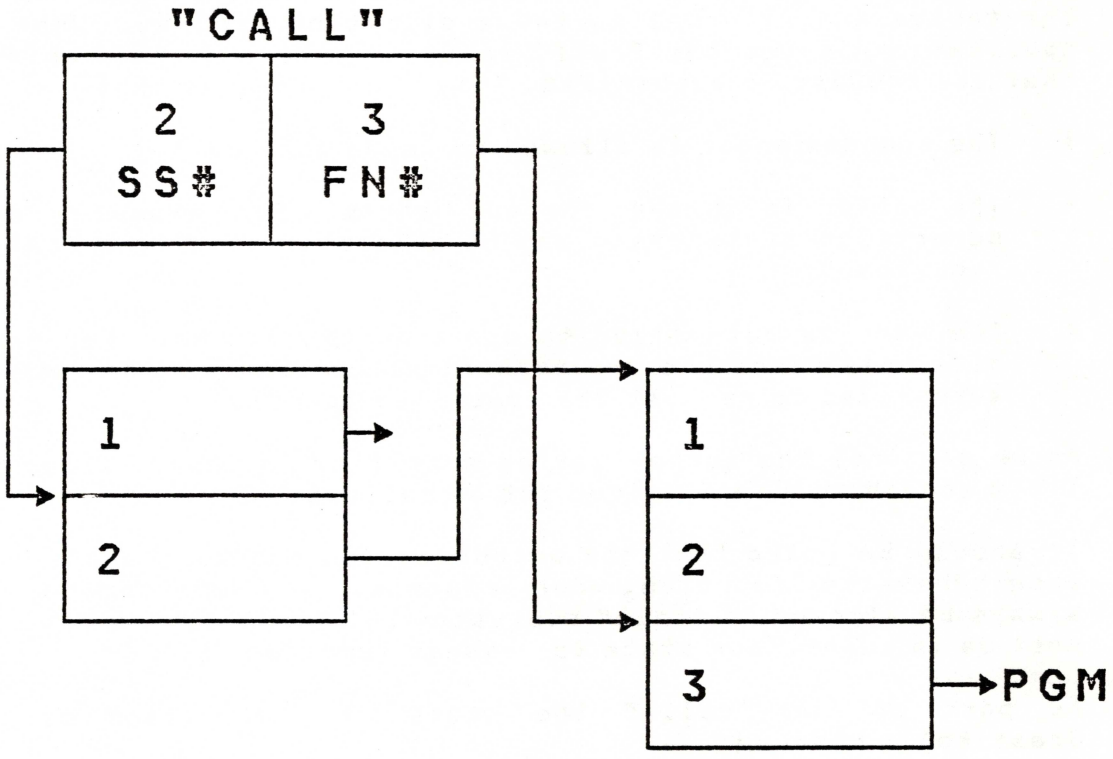
As we can see, the authorization specified in these tables can limit access to a very unique set of callers.

It should be noted that the authorization checking applies to each subsystem function. For example, an application may always be allowed to "call" subsystem SS1 for function "a," but must be in supervisor state to request function "b."

As part of the "call," the state of the caller can be drastically changed:

- Obviously the PSW instruction address (the address of the next instruction to execute) can change.
- The lookup table can specify that the address space is to change. For example, after the "call" the user may be executing instructions in a subsystem's address space. The table can also specify that instructions are to be fetched from the caller's address space (i.e., no address space change takes place). This allows the "call" function to work similar to an SVC instruction (i.e., change from problem program to supervisor state).
- The state can also change across a "call." That is, the PSW can change from problem program to supervisor state.
- After the "call" the user may be authorized to use additional keys. For example, an application only authorized to use key 8 can be authorized to also use key 5.
- The "call" function can also increase the list of address spaces that the application is authorized to update/reference and/or "call." In our previous example (see foil 33 on page 68), after USR2 "calls" SS2 (the data base manager), the application may be authorized to access the associated data base, SS2 DATA.

"CALL" MECHANISM



PROGRAM DESCRIPTION -

ENTRY ADDRESS

ADDRESS SPACE

STATE (PP/SUPR)

KEY(S)

FOIL 36

Here we see a sample "call" request. The "call" function identification consists of two parts:

1. A subsystem number.

   This number must be unique within the system, and is established when the subsystem is initialized.

2. A function number.

   This number is defined by the subsystem.

In our example, the application issues a request for function 3 of subsystem number 2. The system will index into a table to find the "call" table associated with subsystem 2. Using the function code, the system will then index into this table to find the call values associated with this request. After performing the requested authorization checks, control will be passed to the specified routine (which may be in a different address space).

"RETURN" FUNCTION

- RETURNS TO PROGRAM WHICH ISSUES THE "CALL"
- RE-ESTABLISHES CALLER'S ENVIRONMENT

FOIL 37

In conjunction with the "call" function, we must have a "return" function. This mechanism is obviously used to return control to the program which issued the "call." This "return" re-establishes the caller's environment (e.g., state, address space, keys authorized to use). The "return" has the same purpose for cross memory processing as SVC 3 (EXIT) does for SVC processing.

LATE BINDING

- USER SPECIFIES
  - SUBSYSTEM ID
    SHOULD BE EXTERNAL
    TO THE PROGRAM
  - FUNCTION WITHIN
    SUBSYSTEM
- SUBSYSTEM
  - DEFINES FUNCTION
    CODES
  - BUILDS LOOKUP
    TABLES

FOIL 38

The previously described linkage convention moves the information about the called routine from the caller's program to external tables. In fact, the caller is unable to determine the address or name of the called routine. Rather, the user simply specifies the subsystem ID and the function within the subsystem. The subsystem ID should be external to the application. The function number will probably be hard coded in the application program. After all, the application should know what function it wants to perform, but not who will perform it.

The following scenario could be used to determine the subsystem ID:

1. The application program is passed the name of the subsystem (in this case a formal "subsystem") as one of its initialization parameters.

2. The application calls a subroutine which looks up the subsystem name in the Subsystem Communications Vector Table (SSCT) Chain.

3. The associated SSCT contains the subsystem's cross memory ID.

Obviously there are many other techniques which allow the subsystem ID to be external to the application program.

The subsystem must define (and publish) the function codes it supports. It does not have to tell users which modules accomplish the functions. The addresses of the routines must only be specified in the "call" linkage table that the subsystem builds as part of its initialization.

LATE BINDING ADVANTAGES

- REPLACEMENT OF FUNCTION
- TESTING
- ELIMINATE RE-IPL
  - MODULES NOT IN LPA
  - START AFTER IPL
  - RE-START
- ELIMINATE RE-START OF SUBSYSTEM
  - TABLE DRIVEN "CALL" INTERFACE ALLOWS ONLINE MODULE REPLACEMENT

IBM WASHINGTON SYSTEMS CENTER
FOIL 39

FOIL 39

The late binding technique used by the cross memory services linkage interface provides a significant improvement over the interfaces used in most IBM and customer products today. This new interface simplifies the replacement of a function with a new enhanced version of the product. The using application cannot know the names of the routines processing its requests. Similarly, the application can only "call" the subsystem at formally defined entry points. Thus the replacement subsystem does not have to be concerned that applications:

1. Are dependent on the names of the routines to gain control, or

2. Call the subsystem at nonstandard entry points (e.g., at 4 bytes past the defined entry point so that certain registers are not initialized by the called routine).

The late binding also enhances parallel execution of a test and a production version of the subsystem. For example, a production application could use the test version of a subsystem by requesting a different subsystem number in its "call" request.

The "call" linkage also enables subsystems to be modified without a re-IPL of the system. As we have seen, there is no requirement that modules shared among address spaces reside in PLPA (which cannot be modified after IPL). Also, the subsystem can be started after IPL, since the interface tables can be built at any time (as long as it is before other address spaces attempt to use them). This interface also permits new interface tables to be built when the subsystem is re-started.

This late binding technique also supports online module replacement. Today's program linkage conventions (e.g., CALL, LINK) cannot insure that applications have not "bound" themselves to a program. For example, an application may issue a BLDL for a module and use the directory information every time it uses the module. However, the "call" linkage convention insures that an application cannot remember an entry point across calls. If the subsystem updates the "call" linkage tables, it can be sure that all future requests go to the new module.

"MOVE" FUNCTION

- SUPPORTS DATA TRANSFER BETWEEN --
  - DIFFERENT ADDRESS SPACES
  - STORAGE HAVING DIFFERENT PROTECT KEYS
- AUTHORIZATION CHECKING --
  - MOVE ALLOWED
  - KEY(S)
  - STATE

FOIL 40

We will now look at the second portion of inter-address space communication -- data movement between address spaces. Cross memory services support a move:

- Between different address spaces
- Between storage having different protect keys

The move can be from the current address space to an alternate address space or from the alternate to the current address space. Optionally, the move can be within the same address space (to take advantage of the multiple key function).

The "to" and "from" storage can have different protect keys. This could be used, for example, for moving from the user's key 8 storage to protected key 5 storage. Today, movement between these two protect keys requires:

1. Getting the local lock to prevent FREEMAINing of the user's storage.
2. Validating the key of the user's data by switching to the user's key and referencing the data area.
3. Moving the user's data to protected storage (while executing with a PSW key of 0).
4. Releasing the local lock.

Naturally, one does not want to permit all address spaces to perform cross memory data transfers. Therefore the "move" function provides the following authorization checks:

1. Is the address space allowed to access the alternate address space?
2. Is the specified storage key one the user is allowed to use? (See foil 34 on page 72 for information on the key list.)
3. Is the caller in the correct state (problem program or supervisor state)?

The cross memory move function does not have as many authorization checks as does the "call" function. IBM assumes that typically only subsystems will be permitted to move data between address spaces (as opposed to their dependent application programs). Therefore, we will probably only see programs which are:

- Able to move data to all address spaces (e.g., global resource serialization).
- Only able to move data to address spaces using a subsystem "call" (e.g., IMS/VS).
- Unable to move data to other address spaces (application programs).

THIS PAGE INTENTIONALLY LEFT BLANK

DATA SHARING FUNCTION

("ADDRESS" FUNCTION)

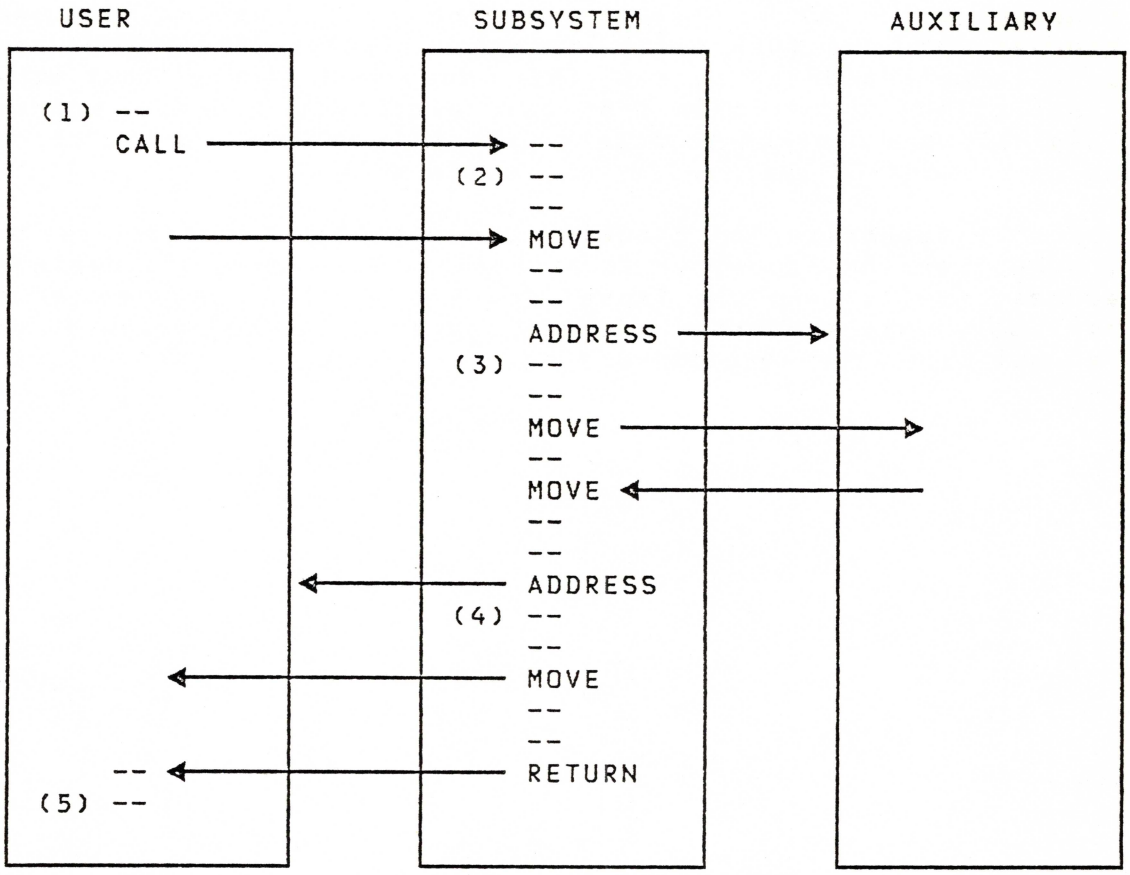
- CAN REQUEST DATA REFERENCE (FETCH/STORE) TO USE ALTERNATE ADDRESS SPACE
- ALLOWS DATA ACCESS TO AUXILIARY ADDRESS SPACE USING "STANDARD" S/370 INSTRUCTIONS
- AUTHORIZATION CHECKING --
  - ADDRESSING ALLOWED

FOIL 41

The third form of inter-address space communication is data sharing. Using cross memory services, an application can request that all future data references (store and fetch) use an alternate address space. This does not imply that the instructions are fetched from this alternate address space. This allows a program to access data in a second address space using "standard" System/370 instructions (e.g., ST, ZAP).

The only authorization checking done by this function is to insure that the requester is authorized to access the desired address space. Like the "move" function, it is assumed that typically only subsystem address spaces will be allowed to request this alternate addressing.

ACCESSING ADDRESS SPACES



(1)	USER	--	--
(2)	USER	SUBSYSTEM	--
(3)	--	SUBSYSTEM	AUXILIARY
(4)	USER	SUBSYSTEM	--
(5)	USER	--	--

IBM WASHINGTON SYSTEMS CENTER
FOIL 42

FOIL 42

This figure is an example of a "typical" multi-address space function. It consists of multiple application address spaces (of which only one is shown), a subsystem address space, and a data only auxiliary address space. The following scenario uses the previously described cross memory services to update and retrieve records in the data base located in the auxiliary address space:

1. The application program "calls" the subsystem. This transfers control to the subsystem address space.

2. The subsystem has access to the application as well as its own address space. Using the cross memory "move" function it copies the input parameter to its own protected workarea.

3. After validating the input, the subsystem establishes addressability to the auxiliary memory. It then updates the data base with the user's input. Next it retrieves the data requested by the application program. Although not shown, this update and retrievable usually involves scanning queues located in the auxiliary address space. This is probably accomplished by temporarily switching data addressing to use this auxiliary memory (data sharing function).

4. After saving the requested data in its memory, the subsystem re-establishes addressability to the user address space. It then moves the data from the subsystem memory to the user's buffer.

5. Processing is now complete. The subsystem "returns" to the calling application program. The application is no longer using cross memory services.

One should not assume that IBM products are using this technique to support cross memory requests. Nor is this an exhaustive example of all of the capabilities of the cross memory extensions.

DISPATCHING

- DISPATCHABLE UNIT

  DOES NOT CHANGE

  - SRB
  - TCB

- CROSS MEMORY USER

  CAN BE SWAPPED

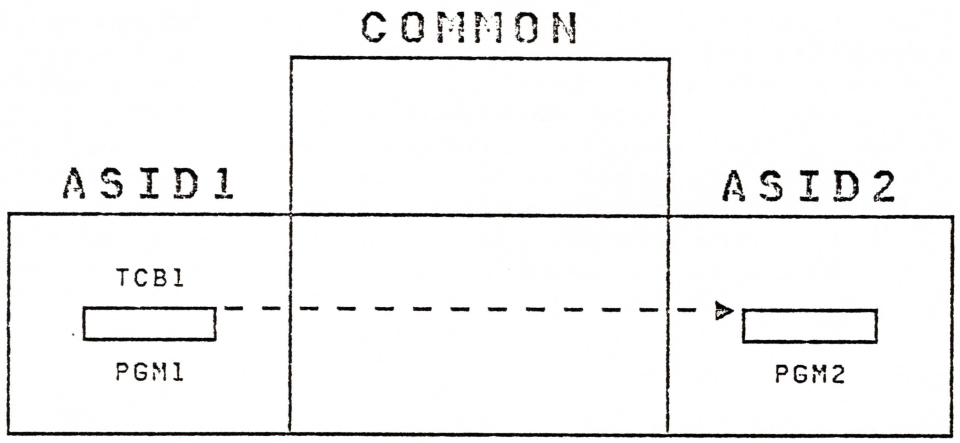
- ACCESSED ADDRESS SPACE

  MUST BE IN STORAGE

FOIL 43

As we have seen, an application executing in one address space can access programs and data in a different address space using the cross memory services provided by MVS/System Product Release 2. However, the dispatchable unit of work (SRB or TCB) does not change. Also, the application program can take page faults or be swapped while executing instructions or accessing data located in this alternate address space. However, the alternate address space must already be swapped in at the time of the access; MVS will not swap in this alternate address space.

DISPATCHING EXAMPLE



1 DISPATCHED
UNDER ASID1

2 CROSS MEMORY
"CALL" ASID2

3 EXECUTE UNDER TCB FOR ASID1 — ADDRESSING IN ASID2

4 SWAP OUT ASID1

5 RESTORE ASID1

6 RESUME PGM EXECUTION — ADDRESSING IN ASID2

FOIL 44

This foil shows an example of the dispatching considerations previously described.

1. PGM1 initially executes in ASID1 under TCB1. This is the "normal" mode of program execution.

2. PGM1 issues a "call" to PGM2 in ASID2. The user must insure that ASID2 has previously been swapped into storage (although the required storage may be paged out). The "call" will fail if ASID2 was swapped out at the time of the request. This typically is not a major restriction since ASID2 usually represents a non-swappable address space such as the IMS control region.

3. The program continues to execute under TCB1 even though it is executing instructions located in the private address space of ASID2. This means that PGM2 is executed at the priority of ASID1. While executing PGM2, page faults may occur accessing common storage (CSA), as well as accessing private storage in either ASID1 or ASID2. ASID2 may not be swapped out as long as other address spaces are accessing its storage. (This probably means ASID2 executes non-swappable). On the other hand, since PGM2 is still executing under TCB1 in ASID1, every access to storage in ASID1 will occur while ASID1 is swapped into storage. Thus ASID1 does not have to execute non-swappable.

4. It is possible for ASID1 to be swapped out because of MPL adjustments or workload manager recommendations. This causes no problems.

5. At some point ASID1 will be swapped back into storage to resume execution.

6. TCB1 resumes executing instructions in ASID2.

SRM/RMF/SMF

- CPU TIME CHARGED TO DISPATCHABLE UNIT
  - TCB
  - SRB
- PAGE FAULTS CHARGED TO ADDRESS SPACE OWNING THE PAGE
- CROSS MEMORY PRIVATE PAGE-IN RATE IS PAGEINS / RESIDENCY TIME
- RMF ARD REPORT WILL INDICATE CROSS MEMORY ADDRESS SPACE

FOIL 45

As one can imagine, several extensions were required to the resource accounting algorithms in MVS/System Product Release 2.

From our previous discussions, we saw that the dispatchable unit of work (TCB or SRB) does not change when an address space accesses an alternate memory. The CPU service continues to be charged to the dispatchable unit of work.

However, page faults are charged to the address space "owning" the page. Thus, when a program executing under TCB1 in ASID1 page faults referencing a page in ASID2, the page fault is charged to ASID2. However, if the page fault occurs referencing a common page, the fault is still charged to ASID1.

The storage isolation function provided by MVS/System Extensions Release 2 allows an installation to protect the response time of an interactive subsystem by specifying a minimum and maximum acceptable page-in rate for its private storage. The page-in rate for an address space is calculated as follows:

```
page_in_rate = #_page_ins / seconds_of_execution_time
```

In MVS/SP R2, this calculation is distorted for address spaces that very seldom execute under their own TCB, but are frequently referenced via cross memory services. These address spaces could have a high number of page-ins but little or no execution time. An example is the JES3AUX address space, which only contains data.

Therefore, in calculating the private area page-in rate for a cross memory address space, the SRM will use residency time rather than execution time. A "cross memory address space" in this case is an address space which can be "called" by other address spaces using cross memory services. This technique treats the private area pages of cross memory address spaces the same as CSA pages are treated today.

RMF[8] Version 2 Release 4 will add a field to its monitor II ARD (Address Space Resource Data) report to indicate whether or not an address space is considered "cross memory." This field would:

- Indicate that the paging rate shown in the PIN RT field (private area page-in rate) was calculated using transaction residency rather than execution time.

---

[8] Resource Measurement Facility, 5740-XY4.

- Indicate that other RMF reports may have unusual values. For example:
  - Very low or zero execution times and EXCP rates.
  - Invalid or distorted AVG RS (average real storage use) values.

THIS PAGE INTENTIONALLY LEFT BLANK

RTM/RAS

- SNAP/ABDUMP
  - CROSS MEMORY CONTROL BLOCKS FORMATTED
  - CROSS MEMORY LINKAGE DATA FORMATTED
- SDUMP
  - CROSS MEMORY DATA DUMPED
  - SUMMARY DUMP DATA
  - STORAGE LIST EXPANDED
    - MORE ENTRIES
    - ASIDS CAN BE SPECIFIED
  - SUSPEND SUMMARY DUMP

FOIL 46

To maintain (and in fact enhance) the system's Reliability, Availability and Serviceability (RAS), significant enhancements were made to RAS components of MVS/System Product Release 2.

When a SYSUDUMP, SYSABEND, or SYSMDUMP is taken, the following additional information will be formatted:

- The control blocks used by cross memory services to determine the types of cross memory access this user is permitted.
- If the user has "called" another address space at the time of the dump, the "call" linkage blocks will be formatted. These blocks are similar to the register 13 save areas that are formatted today.

SDUMP (sometimes known as SVCDUMP or CONSOLE DUMP) has also been enhanced:

- Control blocks that formerly resided in SQA and CSA (e.g., ENQ/DEQ blocks) will be dumped. SDUMP will also include the new cross memory services blocks (e.g., linkage blocks).
- The summary dump information will be cognizant of the cross memory environment. Today, SDUMP outputs 2K bytes around each register and the PSW. With MVS/SP R2, SDUMP will attempt to determine if the address is associated with storage in an alternate address space. SDUMP will then dump the storage from the appropriate memory.
- The storage list option of SDUMP has also been expanded. Besides allowing more entries, the list can specify an ASID with each dump range.
- SDUMP will support a new type of Summary Dump, the "Suspend Summary Dump." As was previously described (foil 26 on page 54), a suspend summary dump is synchronous from the caller's point of view (i.e., the summary data is saved before control is returned to the caller). This means the contents of volatile storage (because the recovery process will modify it before the entire SDUMP is complete) is saved for debugging purposes.

RTM/RAS (CONT)

- SYSTEM TRACE
  - RECORD ADDRESS SPACE IN WHICH EXECUTING
  - RECORD CHANGES IN ADDRESS SPACE ADDRESSIBILITY
- SLIP
  - ADDRESSES CAN SPECIFY ASID QUALIFIERS

FOIL 47

System Trace has also been enhanced to provide additional information for the cross memory services user. With MVS/SP R2, System Trace will:

- Record the address space in which the program is executing (as well as the dispatching address space).

- Record changes in the address space's addressability (e.g., "call," establish addressability to alternate address space).

The SLIP function has been enhanced to allow the DATA, LIST SUMLIST, and TRDATA parameters to allow qualification of each direct or indirect address by an address space identifier.

CROSS MEMORY ENHANCES INTER-ADDRESS SPACE COMMUNICATION

- LESS APPLICATION OVERHEAD (VS. SRB TECHNIQUE)
- LESS COMPLEXITY
- ALLOWS MIGRATION FROM COMMON TO OTHER ADDRESS SPACES

IBM WASHINGTON SYSTEMS CENTER
FOIL 48

FOIL 48

I would now like to spend a final few minutes reviewing the system enhancements provided by MVS/System Product Release 2 cross memory services. Cross memory services provide an inter-address space communication technique which uses minimal system resources. This access requires significantly less application overhead than the SRB scheduling technique currently used. Also, the cross memory environment is less complex than asynchronous SRB processing. Finally, cross memory services allow migration from common to auxiliary address spaces without modifying the application programs.

CROSS MEMORY PROVIDES VIRTUAL STORAGE CONSTRAINT RELIEF

- MOVE DATA FROM COMMON TO AUXILIARY ADDRESS SPACE
- MOVE CODE FROM PLPA TO AUXILIARY ADDRESS SPACE
- HORIZONTAL EXPANSION OF ADDRESSING RANGE
- "INFINITE" VIRTUAL STORAGE

FOIL 49

Since the inter-address space communication problem has been addressed, IBM components and subsystems can now provide virtual storage constraint relief by moving data and modules from common storage to a private address space. This horizontal expansion of storage provides a subsystem with an effectively "infinite" virtual storage addressing range.

CROSS MEMORY ENHANCES SYSTEM & SUBSYSTEM RAS

- "FORCES" BETTER PROGRAMMING
  - PGM INTERFACE EXTERNAL TO USER'S APPLICATION
  - DATA ACCESS VIA "QUERY"
- STORAGE ISOLATION
  - KEY SYSTEM DATA NOT ADDRESSABLE
- ADDRESS SPACE IS UNIT OF PROTECTION
  - NOT KEYS
  - MORE UNITS

IBM WASHINGTON SYSTEMS CENTER
FOIL 50

FOIL 50

The addition of cross memory services also enhances the RAS characteristics of MVS and its subsystems.

By using cross memory services (especially "call") a subsystem can force its user to employ good programming conventions. For example, the interface to a subroutine (e.g., program name, module address) is now external to the application program. This forces the late binding of the user to the subsystem. Also, the subsystem can force the user to access the subsystem only via a "query" function. This insures that the application program is not dependent on the format of the subsystem's control blocks.

By moving data from common (CSA) to an auxiliary private area, sensitive data can be protected against accidental destruction by privileged IBM and customer routines. If the data is not addressable, it cannot be destroyed.

This again makes the address space the unit of protection rather than storage keys. It also obviously provides more units of protection (1,500 versus 15).

CROSS MEMORY ENHANCES AUTHORIZATION CONTROL

- LEVELS OF AUTHORIZATION (VS. APF)
  - VERTICAL
  - HORIZONTAL

FOIL 51

Cross memory services also enhance the authorization mechanisms available with MVS. MVS/SP R2 now provides levels of authorization (versus the single level currently available). These levels can be vertical (e.g., the IMS/VS control region has more authority than its dependent regions) or horizontal (e.g.,the test IMS/VS control region has the same level of authority as the production IMS/VS control region, but they cannot address each other's dependent regions).

CROSS MEMORY PROVIDES FOR MIGRATION SUPPORT

- STORAGE ISOLATION FOR MULTIPLE SUBSYSTEM VERSIONS
- LATE BIND OF PROGRAM TO THE SUBSYSTEM
- MIGRATION OF FUNCTION FROM COMMON TO PRIVATE
- PERMITS "PROTECTION" OF INTERNAL WORKINGS OF THE SUBSYSTEM
  - ACCESS VIA "CALL"

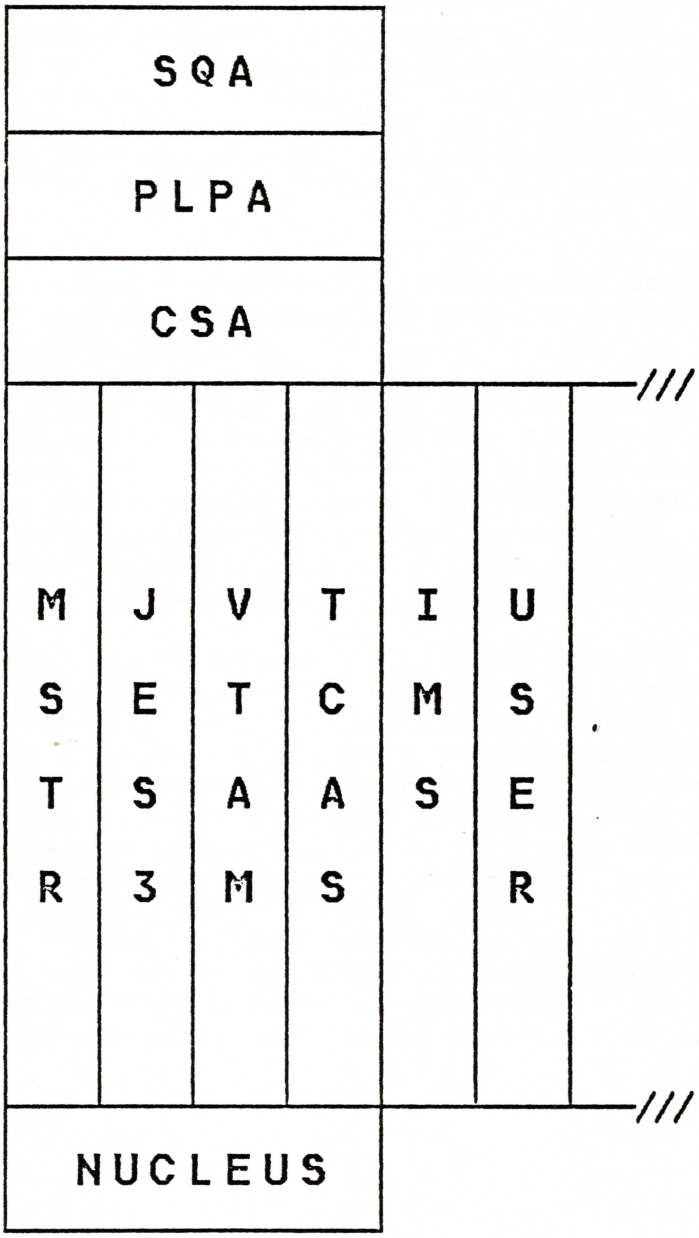
FOIL 52

Cross memory also provides for migration support. The storage isolation previously described permits a test version of the subsystem to safely execute in conjunction with a production version of the subsystem. Since each subsystem is unable to accidentally modify the other's data, there is no concern that the test system might destroy the data associated with the production system. The late binding of the program to the subsystem also eases the testing of an application with a new version of the subsystem.

A subsystem can also use the capabilities of cross memory services to support migration of data and code from common to an auxiliary private area. When an application issues a "call" or addresses data using the cross memory "move," it is unable to tell if the storage is actually in another address space or in common. A previously designed subsystem can simply provide new application interfaces which reference the data and programs still in common storage. At a later time, the subsystem can be modified to support code and/or data in an auxiliary address space. The application will then automatically be switched to accessing the information from the auxiliary private area.

The "call" function can also be used to "protect" the internal workings of a subsystem. By providing a "call" function to access the subsystem's data, the subsystem can be redesigned knowing the application cannot be affected by changes to the control blocks. For example, the ENQ/DEQ rewrite forces customer programs which reference the QCBs and QELs to be modified. However, if global resource serialization is re-written, no customer changes will be required. Global resource serialization will only have to insure that the GQSCAN function returns the same information as it does today, even if the actual control blocks have been re-designed. With the cross memory services implementation, one is assured that an application program cannot be dependent on the format of the control blocks.

CURRENT ADDRESS SPACE STRUCTURE

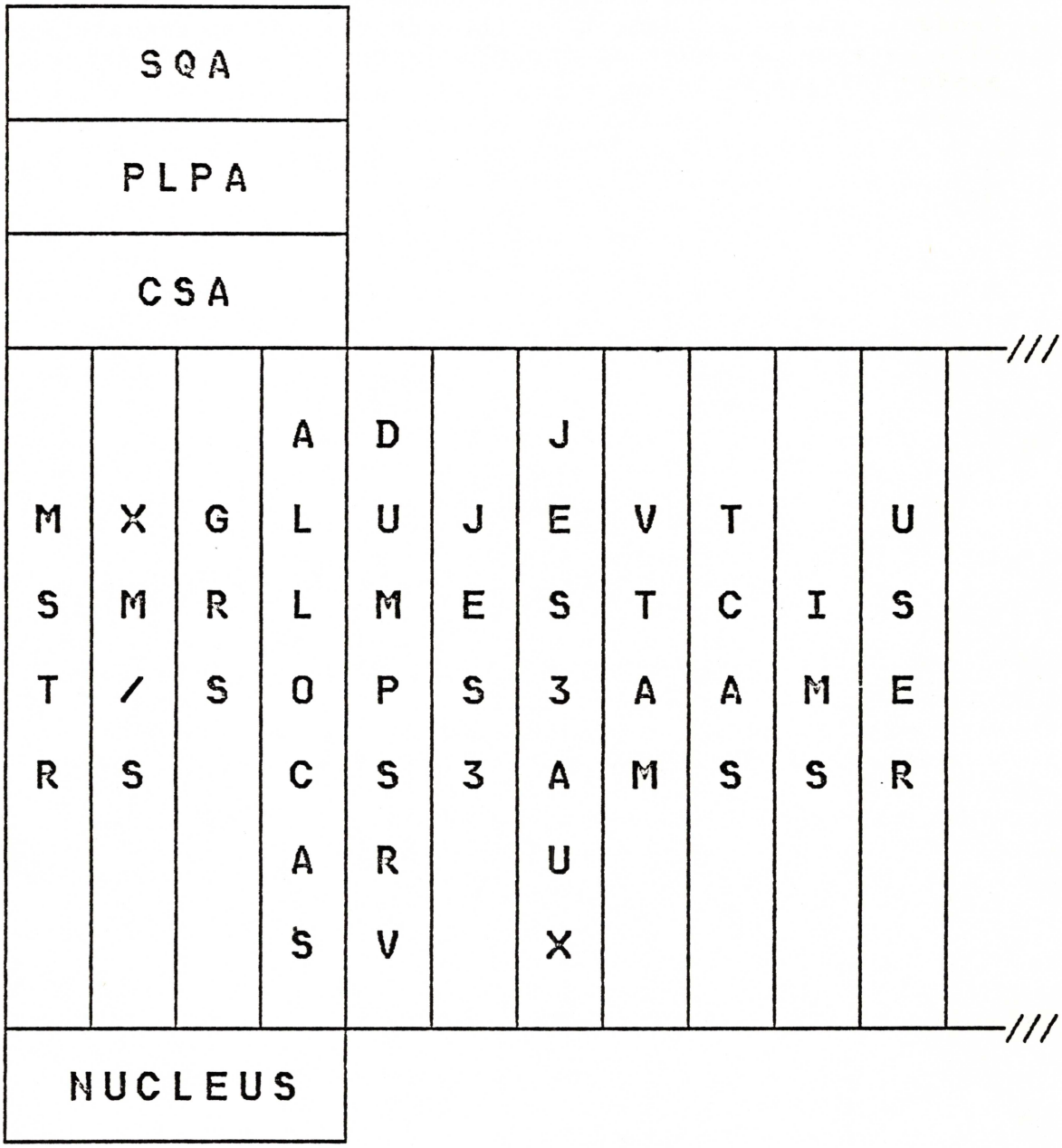


IBM WASHINGTON SYSTEMS CENTER
FOIL 53

FOIL 53

I would like to spend a moment quickly reviewing the MVS/System Product Release 2 changes to the address space structure of MVS. Here we see most of the MVS/System Extensions Release 2 system address spaces.

SP R2 ADDRESS SPACE STRUCTURE



IBM WASHINGTON SYSTEMS CENTER
FOIL 54

FOIL 54

On the other hand, this foil shows the system address spaces added by MVS/System Product-JES3 Release 2. As one can see, MVS has created additional address spaces to contain the new code and data, rather than placing them in commonly addressable storage (i.e., SQA, CSA, and PLPA). Except for the JES3 and JES3AUX address spaces, the same memories have been added by MVS/SP-JES2 R2.

THIS PAGE INTENTIONALLY LEFT BLANK

Reader's Comments

Title: Introduction to Cross Memory Services
Washington Systems Center
Technical Bulletin GG22-9201-00
June 1980

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please state your occupation: _____

Comments:

Please mail to:

P. R. Dorn
IBM Corporation
18100 Frederick Pike
Gaithersburg, MD 20760
U.S.A.

Reader's Comments

Title: Introduction to Cross Memory Services
Washington Systems Center
Technical Bulletin GG22-9201-00
June 1980

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please state your occupation: _____

Comments:

Please mail to:

P. R. Dorn
IBM Corporation
18100 Frederick Pike
Gaithersburg, MD 20760
U.S.A.